6. Introducción al Back-End

6.1. Introducción y objetivos

El *back-end* es la parte de una aplicación web que se encarga, principalmente, de procesar, obtener, almacenar y asegurar todos aquellos recursos o datos necesarios para el correcto funcionamiento del negocio.

Si bien esta definición es correcta, no deja de ser demasiado amplia ya que el *back-end*agrupa una gran cantidad de tecnologías que se encargan de diferentes funciones. Además, una parte clave a la hora de desarrollar un *back-end* es pensar cuál será el mecanismo mediante el cual se posibilite la comunicación entre el mismo *back-end* y el *front-end*. Como veremos más adelante, existen varias formas.

A lo largo de este tema nos centraremos en los siguientes aspectos:

* Qué es exactamente el *back-end* y qué partes de una aplicación o sistema engloba.
* Qué tecnologías se utilizan hoy en día en el *back-end.* Desde lenguajes de programación hasta *frameworks* y gestores de bases de datos.
* Qué son los servicios web y qué es el protocolo SOAP (*Simple Object Access Protocol*).
* Qué es una API REST y por qué son tan utilizadas hoy en día.

6.2. Qué es el back-end

A pesar de que la definición de *back-end* que hemos visto antes es correcta, no deja de ser demasiado amplia y no termina de dejar claro qué componentes de la aplicación están dentro de esta otra cara del desarrollo web.

De hecho, esa es la pregunta que trataremos de contestar. ¿Qué componentes forman parte del *back-end*? Lo primero que debemos tener muy claro es que ningún componente dentro del *back-end* debería ser accesible, alcanzable o modificable por el usuario directamente (quizá sí a través de las *API* —*application programming interface*— o servicios web). Esto es fundamental, ya que nos permite establecer una distinción clara de dónde se encuadra un componente concreto de la arquitectura de una aplicación web.

Dicho esto, los componentes más importantes de un *back-end* son los siguientes:

* Lenguajes de programación orientados al *back-end,* como pueden ser JavaScript, Java, Python o PHP.
* *Frameworks* orientados al *back-end,* que se basan en alguno de los lenguajes indicados anteriormente, como por ejemplo Spring (para Java), Django (para Python) o Laravel (para PHP).
* Bases de datos, ya sean relacionales o no relacionales. La decisión entre usar un tipo u otro depende totalmente del tipo de consultas que la aplicación consumidora requiera y de la necesidad de organizar la información de una u otra forma, teniendo en cuenta siempre la escalabilidad.
* Servidores HTTP (*Hypertext Transfer Protocol*), Servicios web y API REST (*Representational State Transfer*).

Dentro de un *back-end* pueden existir muchísimos más elementos, como servidores de correo, servidores DNS (*Domain Name System*), FTP (*File Transfer Protocol*) o sistemas y gestores de colas (para la mensajería asíncrona entre componentes). Sin embargo, todos estos elementos, aunque pertenecen al *back-end*, estarían gestionados por un equipo de infraestructura y no por un desarrollador web. Para un desarrollador web es fundamental tener conocimiento de los puntos indicados anteriormente.

6.3. Tecnologías principales en el back-end

En el lado del servidor también nos encontramos con tecnologías muy conocidas. En cuanto a los lenguajes de programación, son muchos los lenguajes que se pueden utilizar para construir un back-end o partes de él, aunque se podría decir que destacan los siguientes sobre los demás, debido a la cantidad de recursos disponibles para ellos.

A blue and white text on a blue background

Description automatically generated

A close-up of a text

Description automatically generated

Tabla 1. Lenguajes que se utilizan para programar un *back-end*. Fuente: elaboración propia.

Todos los lenguajes mencionados anteriormente disponen de las herramientas necesarias para comenzar a construir el lado servidor de una aplicación, por lo que la decisión de usar uno u otro depende enteramente de la organización. Tendría mucho sentido conocer PHP si nuestra aplicación web hace uso de CMS y queremos personalizar alguna funcionalidad o añadir otras nuevas, ya que es muy extraño que estas plataformas admitan código en otro lenguaje.

Sin embargo, cada vez es más habitual ver *back-end* escritos con JavaScript, Python o Java, debido a que son lenguajes ampliamente utilizados y, entre otras cosas, es mucho más fácil encontrar desarrolladores con experiencia en ellos.

Observemos a continuación la siguiente gráfica de Stack Overflow que muestra las tecnologías más populares:

A graph with numbers and a number of people

Description automatically generated with medium confidence

Figura 1. Tecnologías más populares. Fuente: StackOverflow, 2021.

Otro aspecto importante para tener en cuenta a la hora de desarrollar un nuevo *back-end*es el almacenamiento de toda la información que la aplicación debe manejar. En las aplicaciones medianamente complejas es necesario disponer de una base de datos en la cual poder almacenar y, posteriormente, consultar toda la información que se necesite. Dependiendo, principalmente, de la forma en la que se quiera almacenar la información y del tipo de consultas a realizar a sobre ella, encontramos varios tipos de bases de datos que pueden estar presentes en un *back-end.*De hecho, es muy habitual encontrar diferentes tipos de bases de datos dentro de un mismo back-end.

Los tipos de bases de datos más comunes son el modelo relacional y el modelo no relacional (NoSQL), aunque existen otros tipos de bases de datos. No obstante, a nivel empresarial, las bases de datos relacionales siguen siendo, por mucho, el tipo de base de datos más utilizado, como podemos ver en la siguiente encuesta de Stack Overflow:

A graph of numbers and text

Description automatically generated with medium confidence

Figura 2. Bases de datos más populares . Fuente: StackOverflow, 2021.

Por último, todo *back-end* debe ser capaz de ofrecer de alguna forma toda la funcionalidad para la que ha sido desarrollado. Es decir, el *front-end* debe ser capaz de comunicarse mediante algún mecanismo con el *back-end* para enviar y recibir datos, lo que habilita al *back-end* para procesarlos de la forma que considere y devolver, posteriormente, un resultado al frontal para mostrarlo al usuario.

En este contexto, las dos opciones más comunes son:

* Utilizar servicios web.
* Exponer una API REST.

Si lo deseas, en la sección A fondo, dentro del recurso *frameworks para lenguajes programación en el back-end*puedes obtener más información sobre los *frameworks* de los que hemos hablado (exceptuando Spring, de Java).

6.4. Servicios web

Tenemos dos subsistemas muy bien diferenciados en una aplicación web, el *front-end* y el *back-end*. Sin embargo, una vez están construidos es imperativo disponer de algún mecanismo que permita la comunicación entre las máquinas de un subsistema y las máquinas del otro. Una forma de conseguir esta integración es a través de un servicio web.

El W3C (*World Wide Web Consortium*) nos da la siguiente definición de este concepto: un servicio web es un sistema software diseñado para soportar la interacción máquina-máquina a través de una red, de forma interoperable. Debe tener una interfaz descrita en un formato preprocesable que permita interactual con el servicio mediante el intercambio de mensajes SOAP.

En efecto, esa es una definición muy acertada de lo que es un servicio web (o *web service*). Para poder utilizar un *web service* es necesario conocer el lenguaje WSDL (*Web Services Description Language*) y el protocolo SOAP, ya que son los dos grandes pilares de este tipo de comunicación.

Lenguaje WSDL

WSDL es un formato de especificación para definir los servicios web basados en XML *(Extensible Markup Language*). Este lenguaje describe cómo es la interfaz pública de un servicio web. Al estar basado en XML, se debe usar dicho lenguaje para componer la descripción de dicha interfaz. Se encarga de describir todos los aspectos de la comunicación, por ejemplo:

* Tipos de datos. Se indican los tipos de datos que se usarán en los mensajes.
* Mensajes. Se indican cómo están compuestos los mensajes a intercambiar.
* Tipos de puerto. Se indican las operaciones permitidas y los mensajes asociados a cada una, su entrada y su salida.
* *Bindings*. Se indican los protocolos de comunicación a utilizar.
* Servicios. Conjuntos de tipos de puerto y dirección.

A screenshot of a computer program

Description automatically generated

Figura 3. Archivo WSDL. Fuente: Tutorials Point, s. f.

Volviendo a los aspectos básicos de la comunicación que ocupa WSDL, para este ejemplo tenemos los siguientes:

* Tipos de datos. Dado que no se va a usar un tipo de dato personalizado y es suficiente con los definidos por XMLSchema, no se aporta ninguna información adicional.
* Mensajes. Se definen dos mensajes,  SayHelloRequest  y  SayHelloResponse . Estos dos mensajes cuentan con un parámetro de tipo *string* en ambos casos.
* Tipos de puerto. Solo se define uno, el correspondiente a la operación  sayHello  que, como vemos, tiene como entrada un mensaje  SayHelloRequest  y como salida un mensaje  SayHelloResponse .
* *Binding.* Nos indica que se debe usar el protocolo SOAP sobre HTTP.
* Servicios. Nos indica la dirección donde este servicio está disponible (<http://www.examples.com/sayHello/>).

Entendiendo esta definición es sencillo realizar llamadas mediante HTTP a cualquier servicio web. No obstante, para poder integrarnos con uno, será siempre un requisito disponer del WSDL donde se detalle como debe ser la comunicación con el servicio.

Además, algunos lenguajes de programación cuentan con librerías capaces de crear todo el esquema de archivos y el código necesario para poder realizar la comunicación con el servicio web.

Protocolo SOAP

SOAP (*Simple Object Access Protocol*) es un protocolo cuyo cometido es definir dos mensajes por medio de intercambio de datos XML. Al igual que ocurría con WSDL, SOAP está basado en XML.

Un mensaje que siga este protocolo puede tener las siguientes partes:

* Sobre (Envelope). Indica qué hay en el mensaje y cómo se debe procesar. Este elemento del mensaje es obligatorio.
* Cabecera (Header). Contiene información relacionada con los nodos SOAP que procesan el mensaje. Este elemento del mensaje es opcional.
* Cuerpo (Body). Ocupado por el contenido del mensaje. Está dirigido al destinatario final del mensaje. Este elemento del mensaje es obligatorio.
* Error (Fault). Se utiliza para notificar errores, si los hay. Este elemento del mensaje es opcional.

A continuación, podemos ver un ejemplo de mensaje siguiendo este protocolo:

A computer code with numbers and letters

Description automatically generated with medium confidence

Figura 4. Mensaje siguiendo el protocolo SOAP. Fuente: elaboración propia.

Los servicios web han sido la forma de comunicación más usada hasta que aparecieron las API REST. Junto con estas, una de las arquitecturas empresariales más famosa, SOA (*Service Oriented Architecture*) utiliza los servicios web como pilar fundamental.

Puedes ver más información de esta arquitectura en el recurso de A fondo, *SOA (Service Oriented Architecture).*

6.5. API REST

REST es un estilo de arquitectura de software que revolucionó el mundo de la ingeniería de software a partir del año 2000. Este estilo fue definido por Roy Fielding, uno de los principales autores de la especificación del protocolo HTTP. Tal es el alcance de este estilo que no existe hoy en día proyecto o aplicación que no haga uso de él a la hora de desarrollar sus API.

Algunas de las características principales de REST son:

* No hay estado.
* Orientación a recursos.
* Uso de métodos HTTP.

REST es stateless

En REST no hay estado. Es *stateless*. Esto quiere decir que, si se ha seguido este estilo al pie de la letra, el servicio perderá todos los datos entre dos llamadas cualesquiera, ya que, al no haber estado, cada petición es única e independiente de cualquier otra. El estado, por lo tanto. es responsabilidad del cliente de la API REST y es el responsable de informar al servicio de este en cada llamada si se considera necesario.

Ejemplo de la característica *stateless* en REST

Un ejemplo de esto podría ser el inicio de sesión. Cuando iniciamos sesión en una aplicación, dicha aplicación nos recuerda hasta que cerramos la sesión, ya sea proactivamente o tras un tiempo determinado de inactividad.

La aplicación de alguna forma nos recuerda y no necesitamos indicarle nuestro usuario y contraseña todo el rato (al menos desde el punto de vista de un usuario).

A la hora de consumir una API REST, será necesario siempre suministrar esas credenciales o algún tipo de token en cada llamada, ya que los recursos no guardarán ninguna información de estado.

El hecho de no tener estado es a la vez una ventaja y una desventaja. Dado que:

* Al ser *stateless* es necesario suministrar constantemente la información de estado. Esto puede ser desmotivador.
* Supone una gran ventaja para la escalabilidad de los servicios, ya que no es necesario disponer de una gran cantidad de memoria para alojar el contenido de cada sesión.

Orientación a recursos

Otro de los puntos clave de REST es la orientación a recursos. En el caso de los servicios web, todo se basa en invocar métodos sobre un servicio remoto (esto se conoce también como RPC o *Remote Procedure Call*), mientras que en REST operamos siempre sobre recursos y no sobre servicios. Estos recursos son accesibles a través de identificadores únicos conocidos como URI (*Uniform Resource Identifier*). Sobre estos recursos se realizarán acciones utilizando los métodos HTTP.

Ejemplo de la orientación a recursos en REST

En el mundo SOAP, podríamos tener un servicio web destinado a la gestión de la flota de un negocio de alquiler de coches. Tendríamos un servicio web llamado CarService con varios métodos como podrían ser AddCar, ModifyCar, RemoveCar o GetCar.

Estos métodos tendrían sus respectivas entradas y salidas y, como hemos dicho, la forma de trabajar con él sería ejecutar métodos (AddCar) de un servicio remoto (CarService).En el caso de REST, la forma de pensar es distinta, ya que ahora no tenemos un servicio, sino un recurso que llamaremos Flota.

Este dispondrá de una URI que lo identificará unívocamente, por ejemplo, /flota. Invocando dicha URI obtendríamos los detalles de esa flota. También podríamos obtener los detalles de un coche concreto dentro de la flota. Sería otro recurso y, por tanto, tendría otra URI asociada, por ejemplo, /flota/123abc, donde 123abc sería el identificador del coche (en este caso su matrícula).

Lo mismo ocurriría con el resto de los métodos definidos en el servicio web. No obstante, para poder identificar si queremos obtener información, añadir, modificar o borrar, es necesario hacer uso de los métodos HTTP.

Métodos HTTP

Una API REST debe hacer uso de los métodos HTTP para indicar el tipo de operación que se va a realizar sobre los recursos. Es decir, es el método HTTP utilizado lo que nos indica qué va a ocurrir, no la URI del recurso.

Hasta que no se introdujo REST, los métodos HTTP estaban infrautilizados. Los más comunes se utilizaban (GET y POST), pero muchos otros estaban totalmente olvidados.

En este contexto, los métodos HTTP más utilizados en una API REST son los siguientes:

* GET. Se usa para operaciones de consulta de datos. Una consulta nunca debería modificar la información de un recurso. Dicho de otra forma, sucesivas llamadas GET a un recurso deberían devolver siempre la misma información (siempre que esta no haya sido modificada por otro lado). Se trata de una operación idempotente.
* POST. Se usa para crear recursos.
* PUT. Se usa para realizar modificaciones sobre recursos.
* PATCH. Se usa para realizar modificaciones parciales sobre recursos.
* DELETE. Se usa para eliminar un recurso.

A la vista de lo anterior, parecería que PUT y PATCH son métodos que realizan lo mismo. Y sí, es cierto, ambos métodos son los encargados de actualizar los recursos. La diferencia está en la cantidad de información que se recibe como entrada de la operación.

Diferencia entre  PUT  y  PATCH

Un coche podría tener muchísima información asociada (matrícula, peso, extras, tipo de ruedas, número de revisiones, kilómetros recorridos, etc.).Si ejecutamos una operación  PUT , deberíamos suministrar un valor para todos los atributos del coche (todos los que hemos mencionado antes). Se usa cuando queremos actualizar el recurso por completo (una sobreescritura total).

Sin embargo, si ejecutamos una operación  PATCH , solo deberíamos suministrar aquella parte del recurso que queramos modificar (solo la matrícula, solo el kilometraje, etc.), por lo que se usará cuando queramos actualizar una parte del recurso.

Es importante recordar que REST es un estilo y, por tanto, es posible encontrar API públicas o privadas donde un método GET sí realice modificaciones en un recurso.

Para que una API sea REST es necesario que tenga todas las características que se han indicado. Lo más común es que las entradas y las salidas de una API REST estén codificadas en JSON. Además, toda petición debe devolver algún código de respuesta HTTP indicativo del resultado de la operación.

Diseñando una API REST

A continuación, nos basaremos en un supuesto para diseñar una API REST aplicando los conceptos expuestos a lo largo del tema.

La empresa de reparto de Madrid,  FastRestFood , está desarrollando una aplicación web y aplicaciones móviles para iOS y Android para que los usuarios puedan utilizar sus servicios. Con el objetivo de dar soporte a los diferentes canales, se ha contratado a una empresa externa que se encargará del desarrollo completo del *back-end.*

Dado que otra empresa será la encargada del desarrollo del *back-end*, se ha acordado compartir una primera versión de una API REST para que el equipo encargado de los frontales pueda progresar en sus desarrollos. La API REST debe soportar las siguientes operaciones desde el punto de vista de negocio:

Creación (1) y consulta de pedidos (todos los disponibles (2) y uno concreto (3))

* Consulta de todos los restaurantes disponibles (4).
* Consulta de un restaurante concreto (5).
* Añadir (6) o eliminar (7) platos a un pedido.
* Añadir (8), modificar (9) o eliminar (10) comentarios en un restaurante.

A partir de ese supuesto, se podría definir una API REST como la siguiente:

A screenshot of a computer error

Description automatically generated

A screenshot of a computer

Description automatically generated

Tabla 2. API REST asociada al supuesto. Fuente: elaboración propia.

La API que se muestra no tiene por qué ser la única posible. Se podrían dar versiones distintas de ella donde los códigos de respuesta o los métodos HTTP utilizados fuesen diferentes.

En la sección de A fondo dispones de más información sobre los códigos de respuesta HTTP. Códigos de respuesta HTTP

A continuación, comentaremos algunos aspectos de la API propuesta.

Se ha decidido utilizar  PUT  para la modificación de comentarios y  PATCH  para añadir y eliminar platos de un pedido. ¿Por qué? La respuesta a esto no es absoluta. Lo primero que debemos pensar es ¿tiene sentido una operación  PATCH  para añadir o eliminar? La respuesta es que depende del recurso y la información tratada.

En este caso queremos eliminar platos asociados a un recurso, pero no queremos eliminar los platos ni el pedido, solo queremos modificar la composición del pedido. Además, lo más probable es que estas operaciones se realicen de una en una, por lo que serán pequeñas actualizaciones parciales al recurso. Parece que podría encajar con una operación  PATCH .

En el caso de los comentarios, los atributos del recurso que se pueden modificar son muy escasos (solo el propio texto del comentario y la valoración). Quizá sea probable que una modificación del comentario lleve cambios en ambos atributos. Ya que se va a modificar toda la información del recurso comentario, tiene sentido utilizar un  PUT .

Muchos códigos de respuesta son compartidos. Es decir, hay algunos códigos de respuesta como 400, 404 o 500 que deberían poder ser devueltos por prácticamente cualquier operación, ya que son situaciones posibles que podrían llegar a darse.

Los mensajes de aceptación difieren si se trata de una consulta o actualización (código 200) o de una creación de un recurso (201). Esto es debido a que se usa el código 201 específicamente para indicar que un nuevo recurso fue creado, además de que la operación terminó exitosamente.